Proyecto Final: Compilador para MiniLang

Logotipo

Descripción generada automáticamente

Alberto Moreno Alcácer y Ferran Bleda Fabado

Grado de Ingeniería Informática – Lenguajes de Programación

**1. Descripción de MiniLang**

MiniLang es un lenguaje de programación diseñado para este proyecto. Es un lenguaje simple orientado a subrutinas que admite variables, operaciones aritméticas y de concatenación, estructuras de control (como bucles while), y llamadas a subrutinas. Cada programa de MiniLang debe consistir en una o más subrutinas.

**Características principales de MiniLang:**

* **Subrutinas:** Declaradas usando la palabra clave SUBROUTINE. Cada subrutina debe especificar su tipo de retorno (por ejemplo, int, void).
* **Variables:** Soporte para tipos básicos como int y string.
* **Estructuras de control:** Soporte para bucles while.
* **Operaciones:** Aritmética básica (+, -, \*, /) y concatenación de cadenas usando el operador +.
* **Impresión:** Uso de la función print para mostrar valores.

Ejemplo de programa:

SUBROUTINE int greet(string name) DO

print("Hola, " + name + "!");

return 0;

END

SUBROUTINE void main() DO

greet("Juan");

END

**2. Instrucciones para compilar y ejecutar**

**Requisitos previos:**

* Python 3.11 o superior.
* Instalación del compilador en una carpeta organizada con los archivos de código fuente.

**Estructura del proyecto:**

* pl0/ (directorio principal del compilador):
  + Archivos principales como semantic\_analysis.py, lexical.py, parse.py, etc.
* Archivos de prueba (test1.ml, test2.ml, test3.ml).

**Pasos para ejecutar el compilador:**

1. Abre una terminal y navega al directorio principal del proyecto.
2. Ejecuta el siguiente comando para analizar un programa MiniLang:
3. python -m pl0 test1.ml -a semantics

Reemplaza test1.ml con el nombre del archivo que deseas probar.

1. El compilador generará tokens, un AST (Abstract Syntax Tree) y realizará el análisis semántico del programa.

**3. Programas de prueba**

**test1.ml**

SUBROUTINE int greet(string name) DO

print("Hola, " + name + "!");

return 0;

END

SUBROUTINE int pow(int base, int exponent)

int pow = 1;

DO

WHILE exponent > 0 DO

pow = pow \* base;

exponent = exponent - 1;

END

return pow;

END

SUBROUTINE void main() DO

greet("Juan");

END

**test2.ml**

SUBROUTINE int greet(string name) DO

print("Hola, " + name + "!");

return 0;

END

SUBROUTINE void main() DO

greet("Juan");

END

**test3.ml**

SUBROUTINE int pow(int base, int exponent)

int pow = 1;

SUBROUTINE int MULT(int a, int b) DO

return a \* b;

END;

DO

WHILE exponent > 0 DO

pow = MULT(pow, base);

exponent = exponent - 1;

END

return pow;

END

**4. Asignación de tareas**

Para la realización de este trabajo, Alberto Moreno se encargó del desarrollo del lexer y el parser y Ferran Bleda del semántico y de la resolución de errores que surgieron a última hora en el sintáctico.

**5. Uso de IA**

Nos hemos apoyado en el uso de ChatGPT para la resolución de un error en la parte del sintáctico, ya que nos daba un error que no sabíamos solucionar y esta herramienta supo solventarlo.

**6. Informe final**

**Flujo general del compilador:**

El compilador está dividido en varias etapas:

1. **Análisis léxico:** Convierte el código fuente en tokens.
2. **Análisis sintáctico:** Construye un AST basado en la gramática de MiniLang.
3. **Análisis semántico:** Verifica la validez del programa, asegurándose de que no existan errores como reasignaciones de constantes, tipos incompatibles o llamadas a funciones no declaradas.

**Decisiones técnicas importantes:**

* **Diseño modular:** Cada etapa del compilador (léxico, sintaxis, semántica) se implementó en archivos separados para facilitar la extensión y el mantenimiento del código.
* **Tabla de símbolos:** Una estructura clave que asegura la correcta gestión de variables y subrutinas dentro de los ámbitos locales y globales.

**Desafíos enfrentados:**

1. **Gestión de ámbitos locales:** Se requirió un diseño cuidadoso de las tablas de símbolos para diferenciar correctamente entre variables locales y globales.
2. **Errores semánticos complejos:** Implementar verificaciones como la correspondencia de tipos y el control de constantes fue desafiante debido a la necesidad de recorrer y validar el AST en profundidad.
3. **Concatenación de cadenas:** Garantizar que las operaciones entre tipos heterogéneos (como int y string) fueran manejadas correctamente.

**Conclusión**

El proyecto logra implementar un compilador funcional para MiniLang con soporte para subrutinas, variables, y estructuras de control. Se destaca la organización modular del código y la inclusión de un análisis semántico robusto que asegura la validez de los programas. A pesar de los desafíos, las decisiones técnicas adoptadas resultaron en un compilador eficiente y fácil de entender.